МІНІСТЕРСТВО ОСВІТИ I НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ БІОМЕДИЧНОЇ ІНЖЕНЕРІЇ

КАФЕДРА БІОМЕДИЧНОЇ КІБЕРНЕТИКИ

**Комп’ютерний практикум №2**

з дисципліни «**Нейронні мережі**»

на тему: «МЕТОДИ ОПТИМІЗАЦІЇ»

**Виконав:**

студент гр. БС-03

Затуловський Г. А.

**Перевірив:**

ас. каф. БМК Дюмін О.Д.

Зараховано від \_\_\_.\_\_\_.\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис викладача)

Київ-2023

# Варіант 6

**Мета роботи:** ознайомитися із різноманітні методи оптимізації, які використовуються в машинному навчанні, зокрема, в області нейронних мереж.

# Практична частина

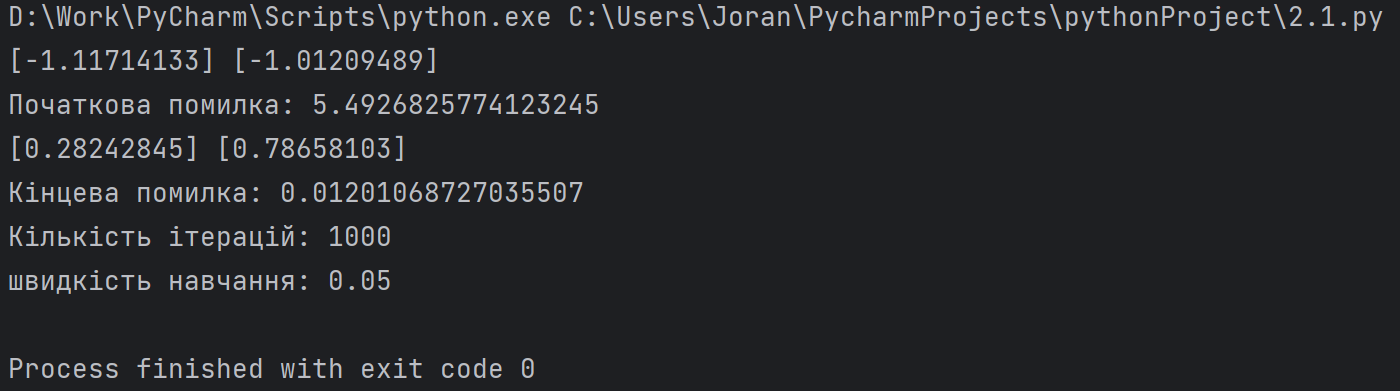
Виконати завдання трьома способами: градієнтним спуском, методом Hill Climbing, генетичним алгоритмом. Проаналізувати ефективність роботи кожного з алгоритмів: отримана точність, кількість ітерацій для отримання бажаної точності, необхідні ресурси, тощо.

**[6, 13, 20, 27] Степенева функція: y = x0.5 + noise**

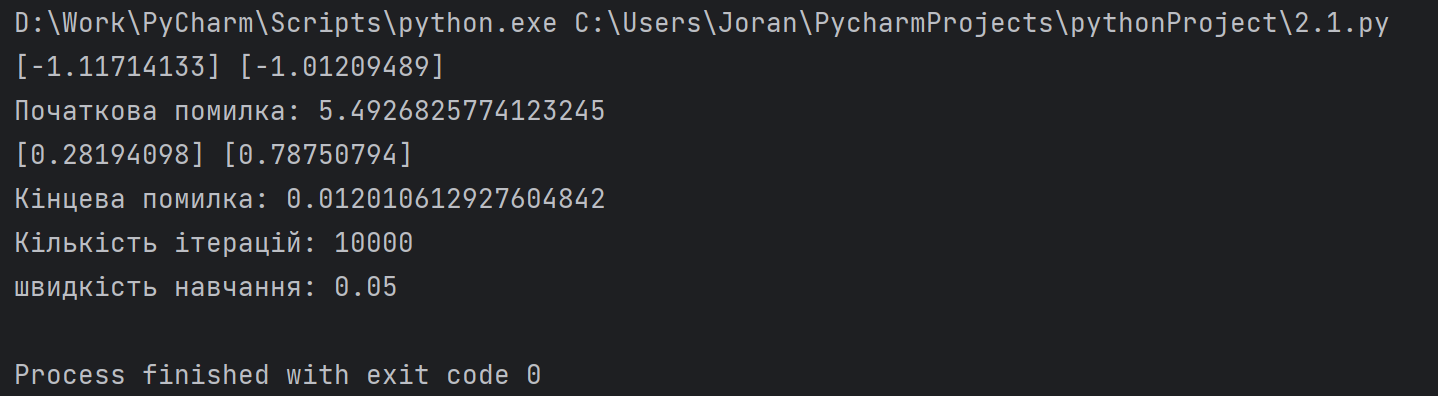
"noise" – це деякий випадковий шум, який ви додаєте до функції, щоб зробити дані менш ідеальними і більш схожими на реальні дані, з якими ви могли б зустрітися в практичних ситуаціях. Наприклад, ви можете використовувати np.random.normal(loc=0.0, scale=0.1, size=x.shape) для додавання гаусового шуму з середнім значенням 0 і стандартним відхиленням 0.1.

1. **Градієнтний спуск**
2. import numpy as np  
   import matplotlib.pyplot as plt  
   # ініціалізація послідовності випадкових чисел  
   np.random.seed(42)  
   # створюємо np-масив з 1000 випадкових чисел в діапазоні 0..1  
   sz = 1000  
   x = np.random.rand(sz, 1)  
   # будуємо функцію y = f(x) та додаємо трохи гаусового шуму  
   y = x\*\*(1/2) + np.random.normal(loc=0.0, scale=0.1, size=x.shape)  
     
     
   # формуємо індекси від 0 до 999  
   idx = np.arange(sz)  
   # випадково перемішуємо  
   np.random.shuffle(idx)  
   train\_idx = idx  
   # формуємо набори навчальних даних  
   x\_train, y\_train = x[train\_idx], y[train\_idx]  
     
   plt.plot(x\_train, y\_train)  
   plt.axis([0, 1 , 0, 1])  
   plt.show()  
     
   # Встановлюємо початкові випадкові значення коефіцієнтів лінійної регресії  
   a = np.random.randn(1)  
   b = np.random.randn(1)  
   print(a, b)  
   # розрахунок початкової помилки (середньоквадратична функція помилки)  
   initial\_error = ((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
     
   print(f"Початкова помилка: {initial\_error}")  
   # швидкість навчання  
   lr = 0.1  
   # кількість епох  
   n\_epochs = 1000  
   # основний цикл  
   for epoch in range(n\_epochs):  
    # обчислюємо отриманий масив з коефіцієнтами A і B  
    # На основі тренувального зразка  
    yhat = a + b \* x\_train  
     
    # 1. Визначаємо втрати  
    # розглядаємо відхилення нового результату:  
    error = (y\_train - yhat)  
     
    # 2. Рахуємо градієнти (за формулою похідної)  
    # для коефіцієнта a  
    a\_grad = -2 \* error.mean()  
    # для коефіцієнта b  
    b\_grad = -2 \* (x\_train \* error).mean()  
     
    # 3. оновлюємо параметри, використовуючи коефіцієнт швидкості навчання  
    old\_error =((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
    a = a - lr \* a\_grad  
    b = b - lr \* b\_grad  
     
   print(a, b)  
   # розрахунок помилки після оптимізації  
   final\_error = ((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
     
   print(f"Кінцева помилка: {final\_error}")  
     
   print(f"Кількість ітерацій: {n\_epochs}")  
   print(f"швидкість навчання: {lr}")

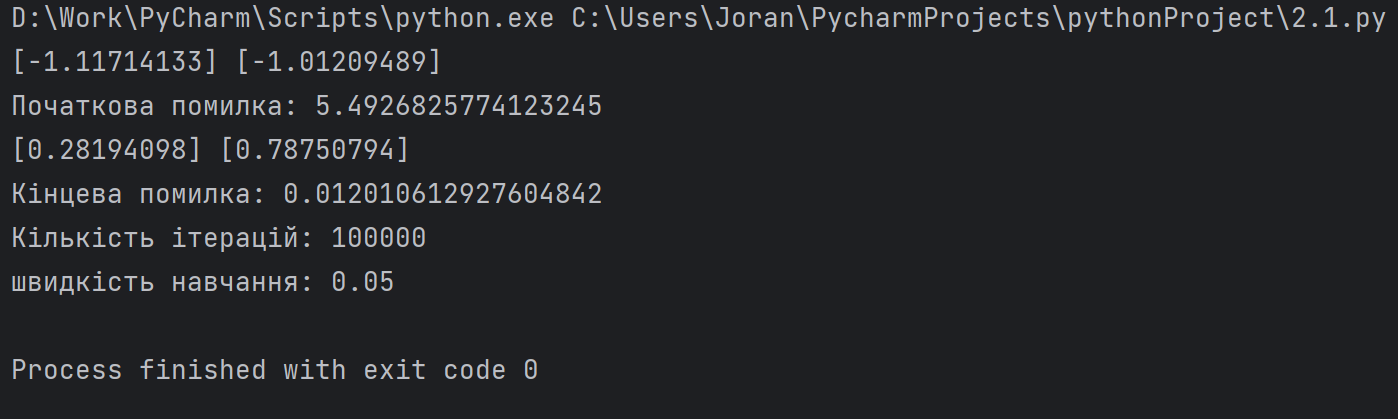
Для 1000 ітерацій, швидкість навчання 0.05



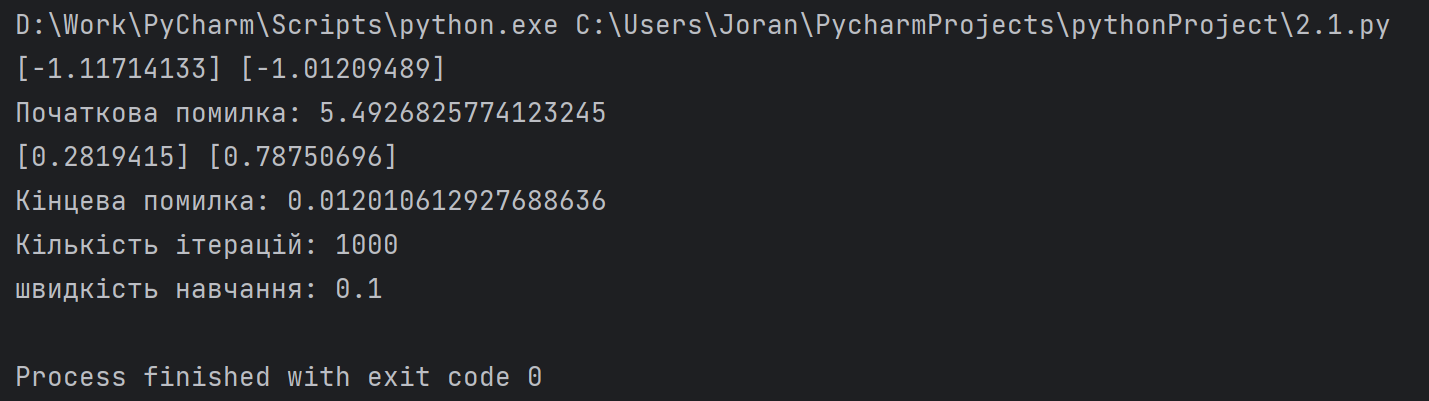
Для 10000 ітерацій, швидкість навчання 0.05



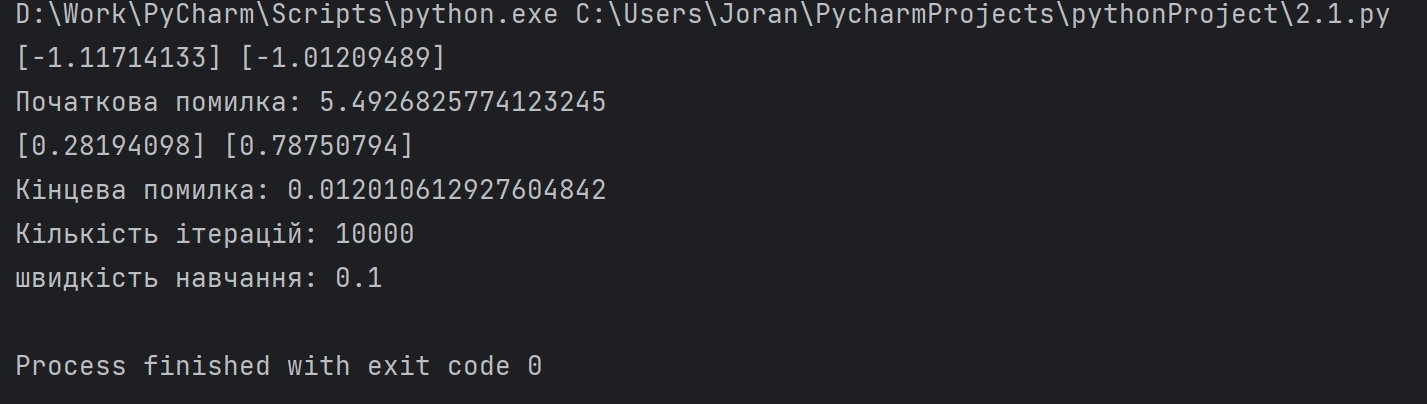
Для 100000 ітерацій, швидкість навчання 0.05



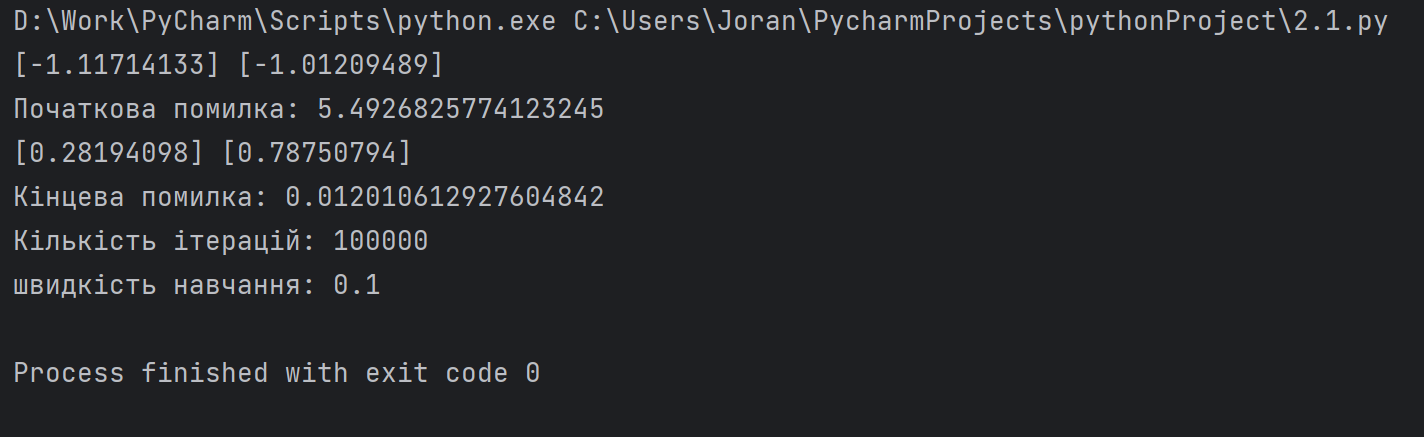
Для 1000 ітерацій, швидкість навчання 0.1



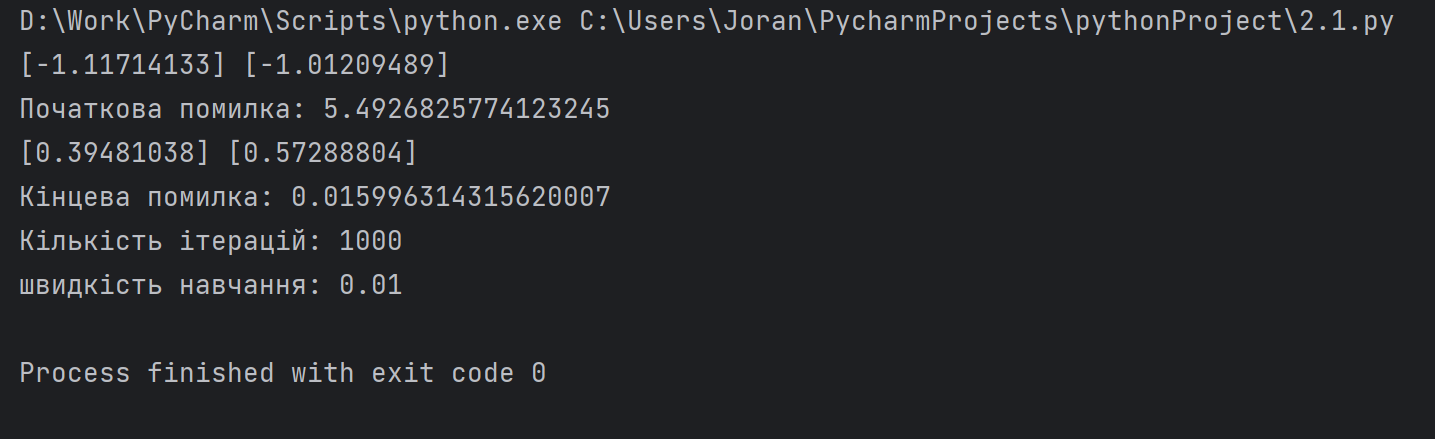
Для 10000 ітерацій, швидкість навчання 0.1



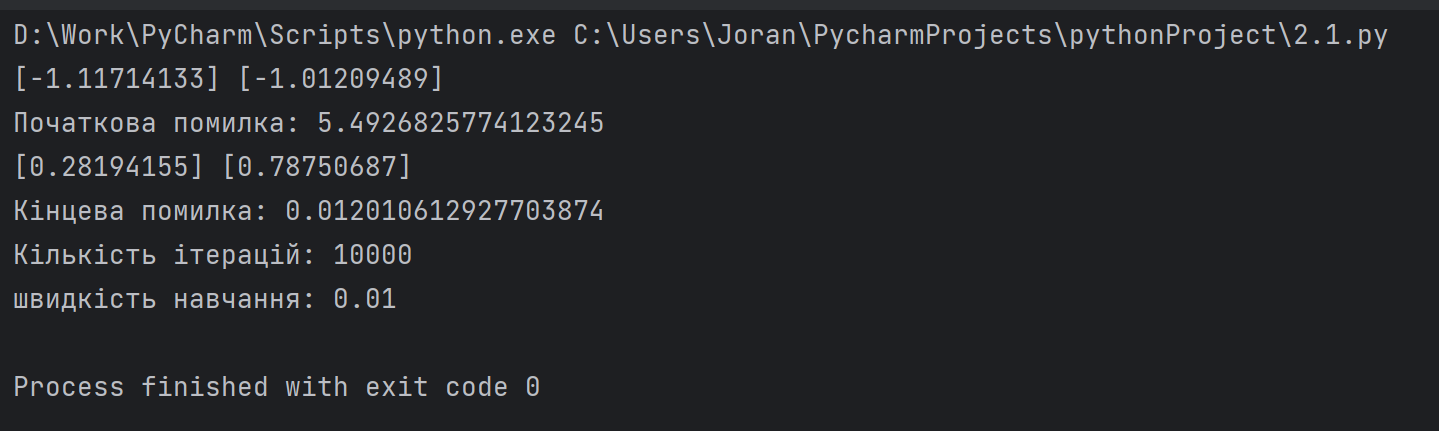
Для 100000 ітерацій, швидкість навчання 0.1



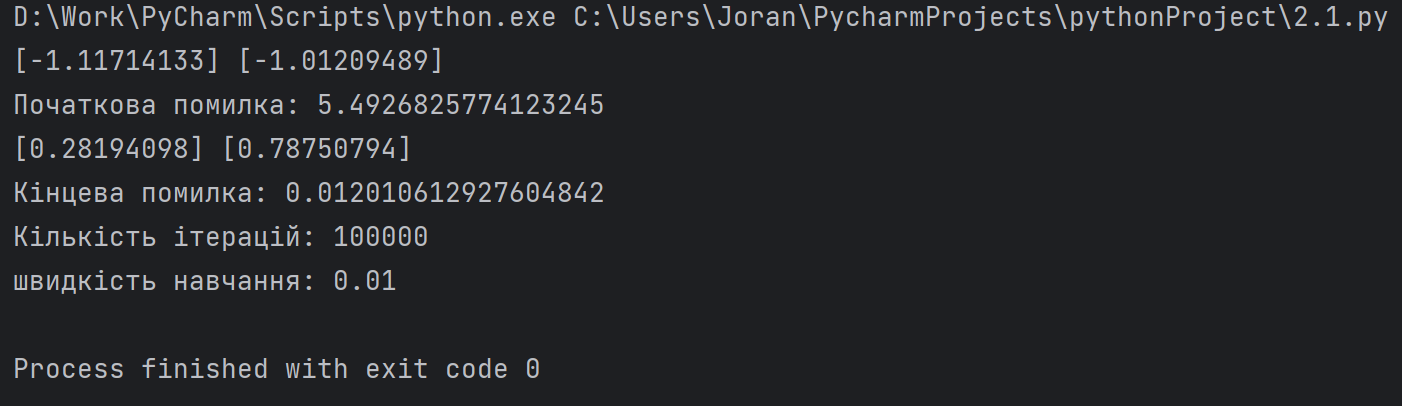
Для 1000 ітерацій, швидкість навчання 0.01



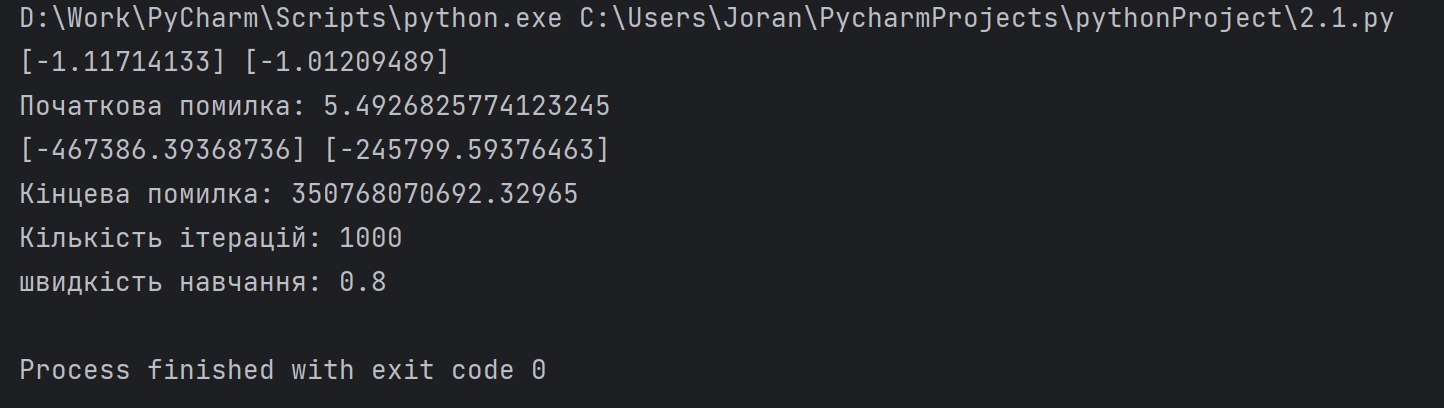
Для 10000 ітерацій, швидкість навчання 0.01



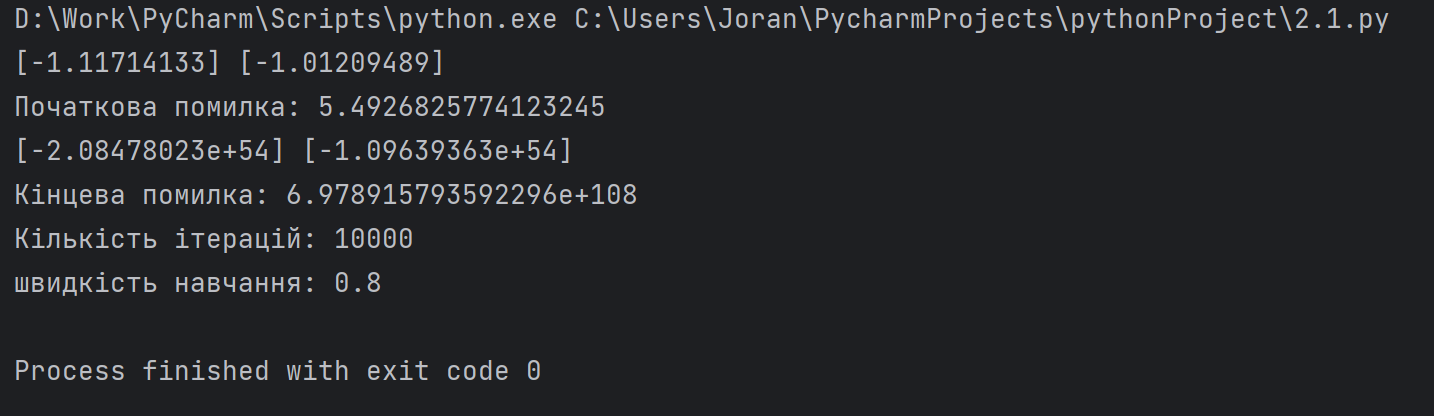
Для 100000 ітерацій, швидкість навчання 0.01



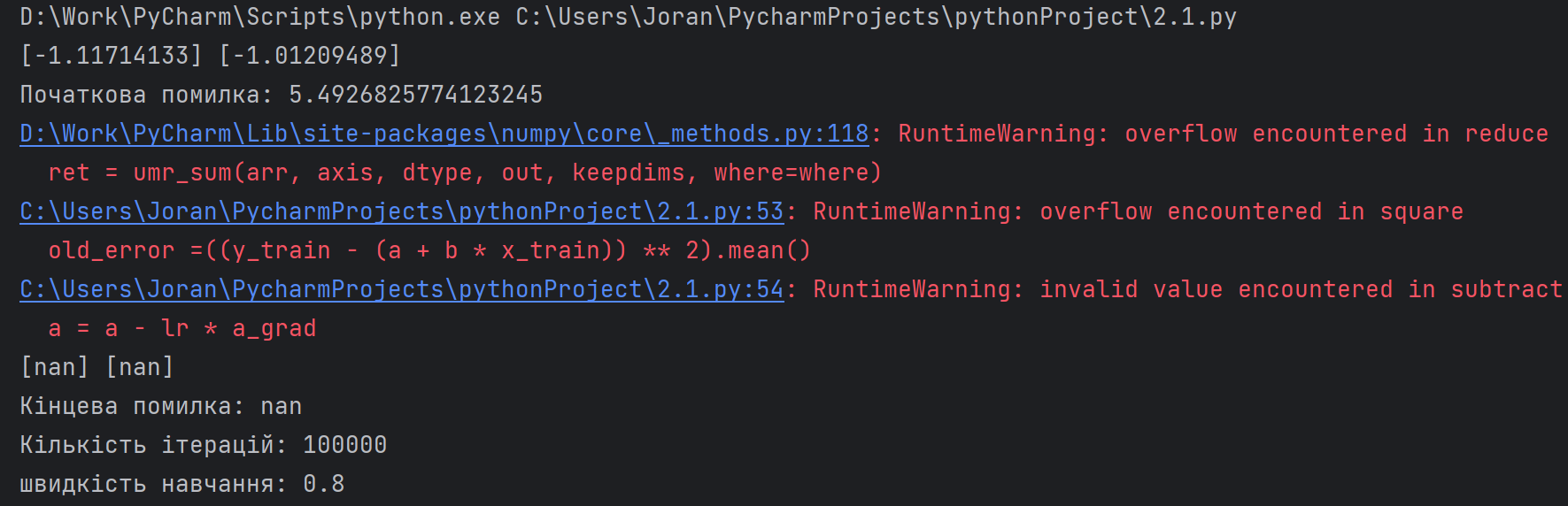
Для 100000 ітерацій, швидкість навчання 0.8



Для 100000 ітерацій, швидкість навчання 0.8



Для 100000 ітерацій, швидкість навчання 0.8

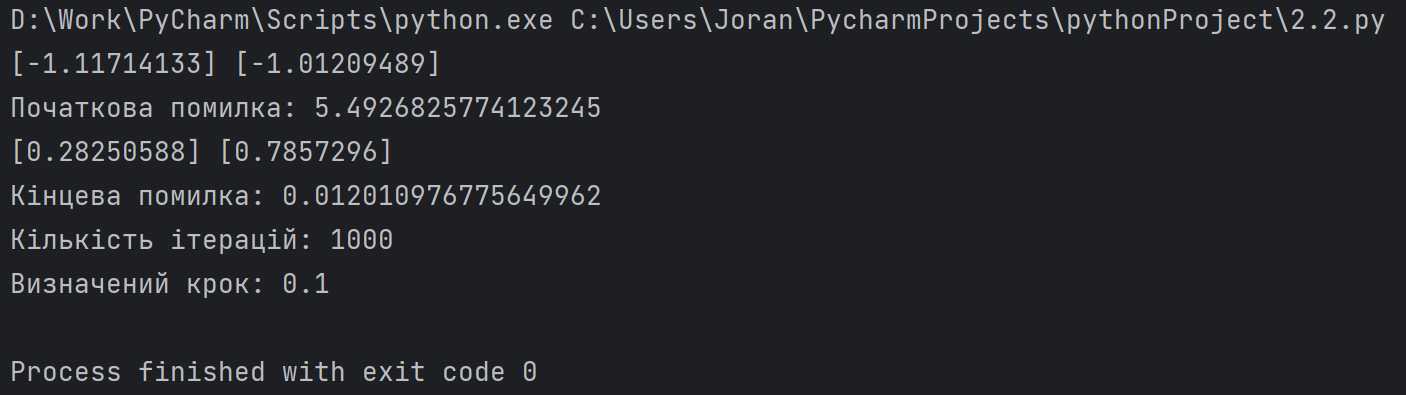


Отже, зі збільшенням кількості ітерацій та швидкістю навчання зменшується кінцева помилка, але при занадто великому значенні (у нашому випадку це 0.8) швидкості навчання моделі, оптимізація починає некоректно оптимизувати наші дані. Найкращий результат оптимізації 0.012010612927604842, який був отриманий при ітераціею 100000 та швидкістю 0.01, 0.1, 0.05

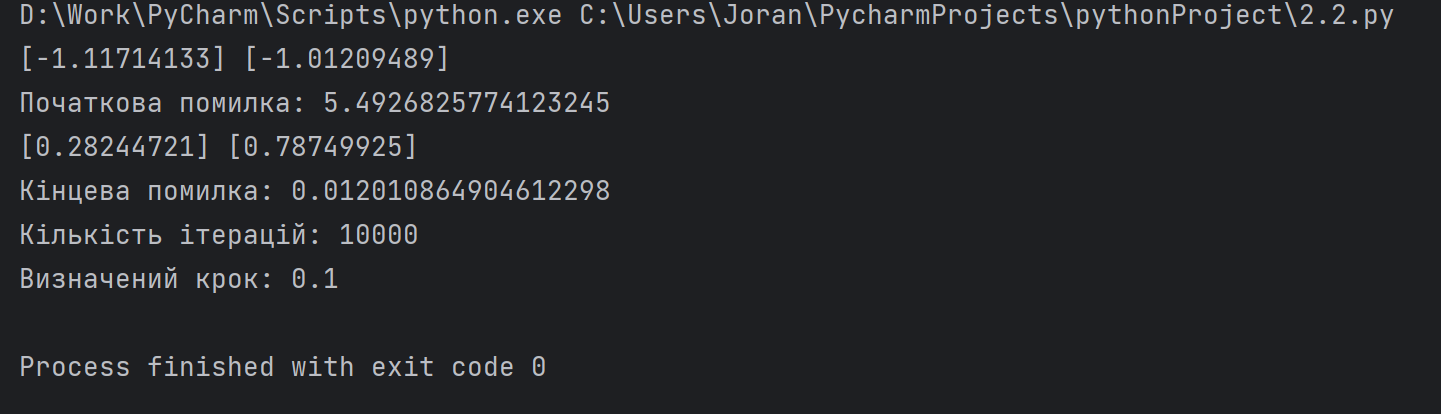
1. **Метод Hill Climbing**

import numpy as np  
np.random.seed(42)  
sz = 1000  
x = np.random.rand(sz, 1)  
y = x\*\*(1/2) + np.random.normal(loc=0.0, scale=0.1, size=x.shape)  
idx = np.arange(sz)  
np.random.shuffle(idx)  
x\_train, y\_train = x[idx], y[idx]  
# Задаємо початкові параметри  
a = np.random.randn(1)  
b = np.random.randn(1)  
print(a,b)  
# розрахунок помилки на початку  
initial\_error = ((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
print(f"Початкова помилка: {initial\_error }")  
# Визначаємо крок (можливо, потрібно налаштувати)  
step\_size = 0.05  
# Визначаємо кількість ітерацій  
num\_iters = 100000  
for \_ in range(num\_iters):  
 # Спробуємо змінити 'a' трохи  
 a\_new = a + (np.random.rand() - 0.5) \* step\_size  
 b\_new = b + (np.random.rand() - 0.5) \* step\_size  
 # Обчислюємо помилку для старого та нового 'a'  
 old\_error = ((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
 new\_error = ((y\_train - (a\_new + b\_new \* x\_train)) \*\* 2).mean()  
 # Якщо нова помилка менша, оновлюємо 'a' та 'b'  
 if new\_error < old\_error:  
 a, b = a\_new, b\_new  
  
print(a, b)  
# розрахунок помилки після оптимізації  
final\_error = ((y\_train - (a + b \* x\_train)) \*\* 2).mean()  
print(f"Кінцева помилка: {final\_error}")  
print(f"Кількість ітерацій: {num\_iters}")  
print(f"Визначений крок: {step\_size}")

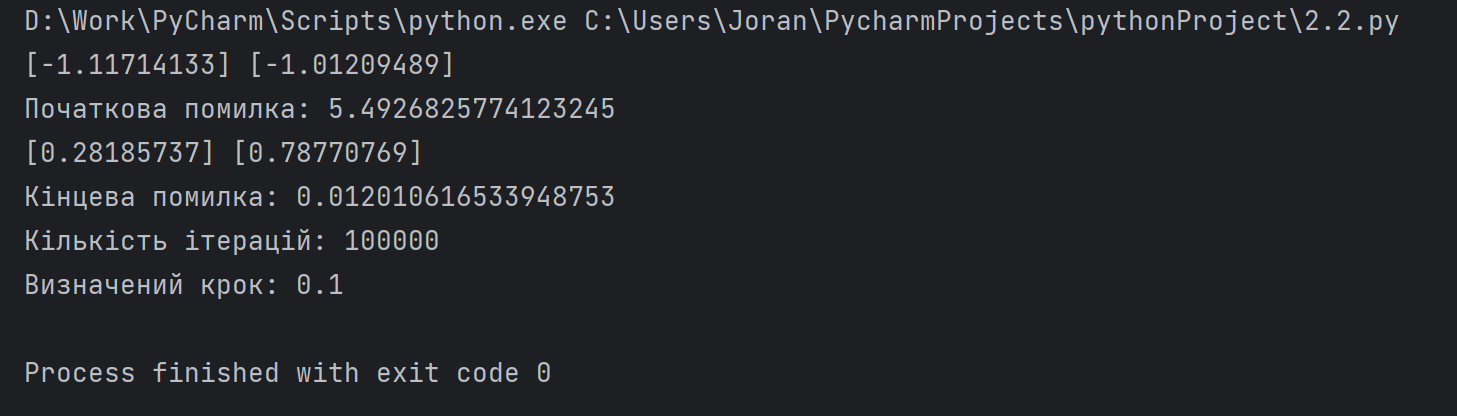
Для 1000 ітерацій, швидкість навчання 0.1



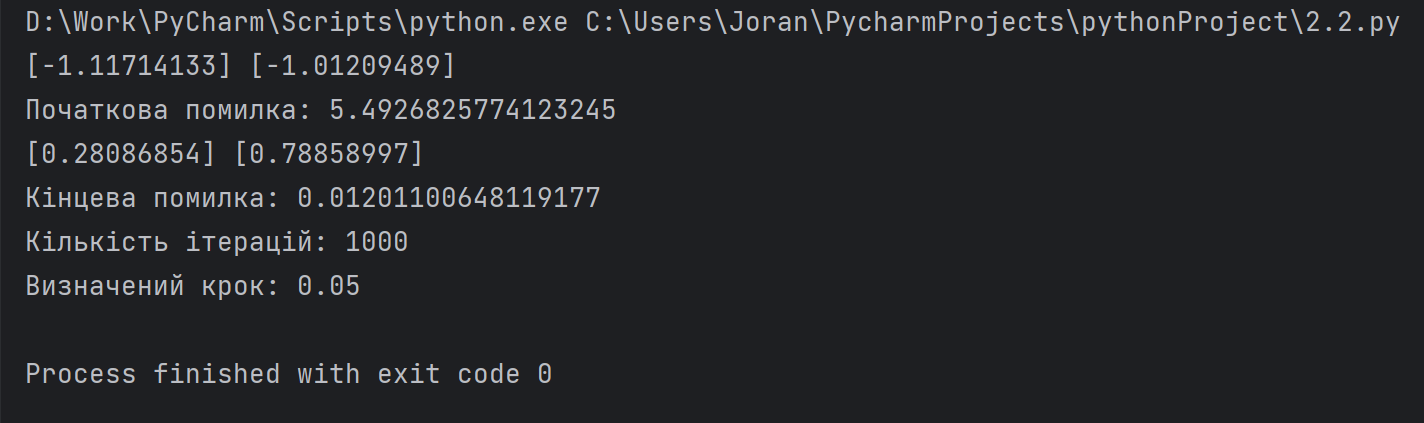
Для 10000 ітерацій, швидкість навчання 0.1



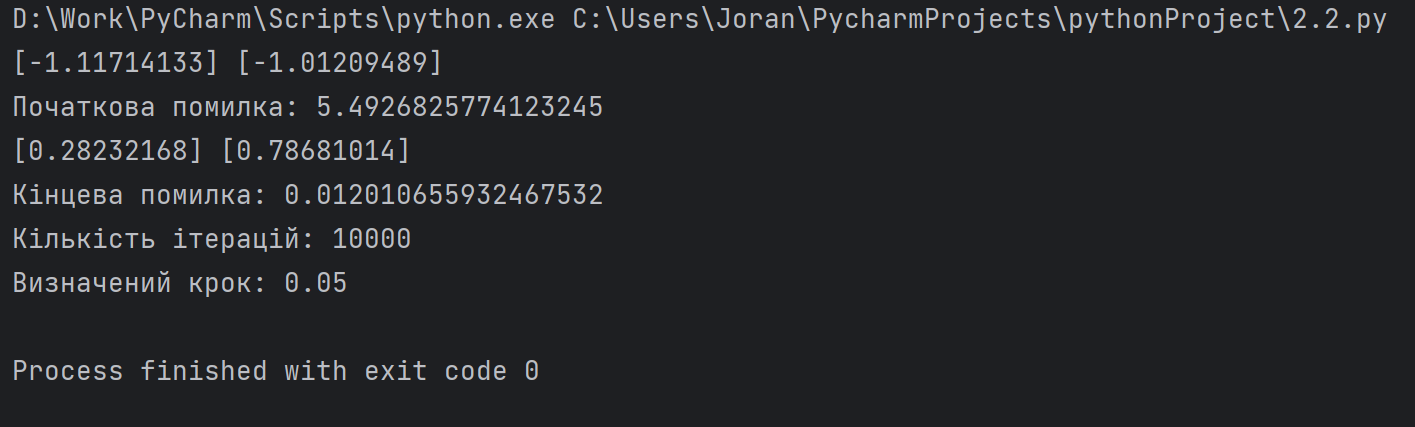
Для 100000 ітерацій, швидкість навчання 0.1



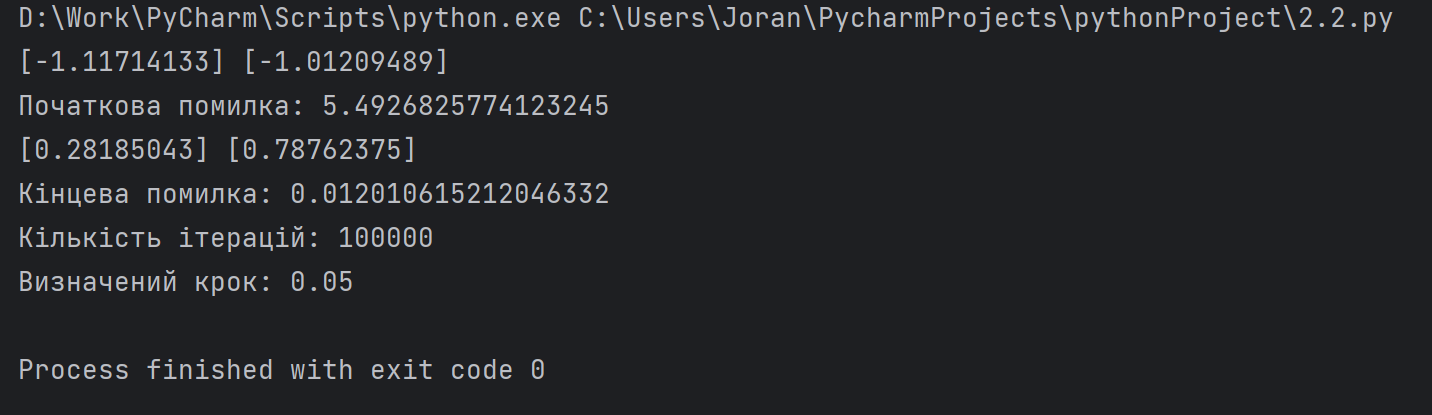
Для 1000 ітерацій, швидкість навчання 0.05



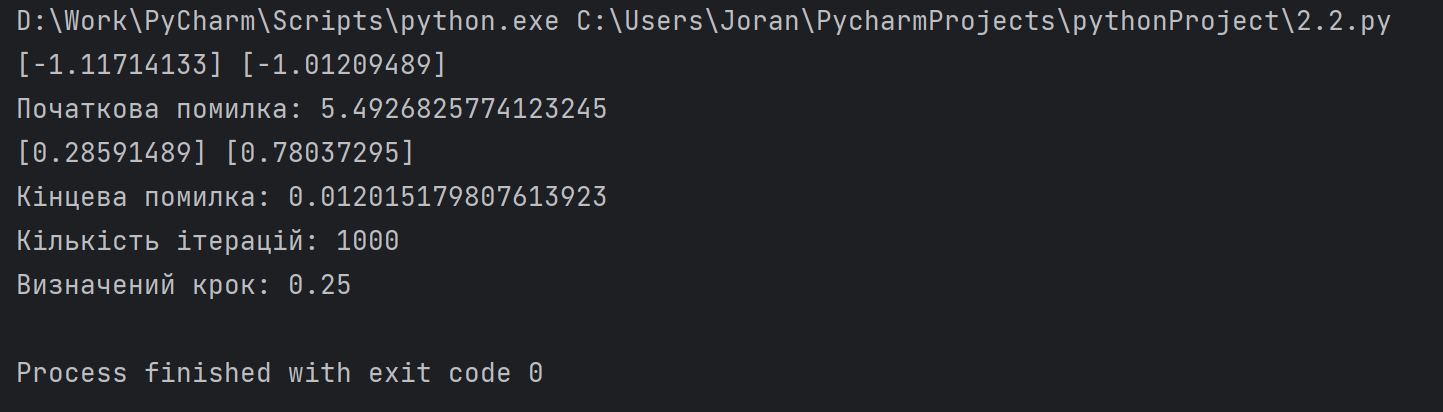
Для 10000 ітерацій, швидкість навчання 0.05



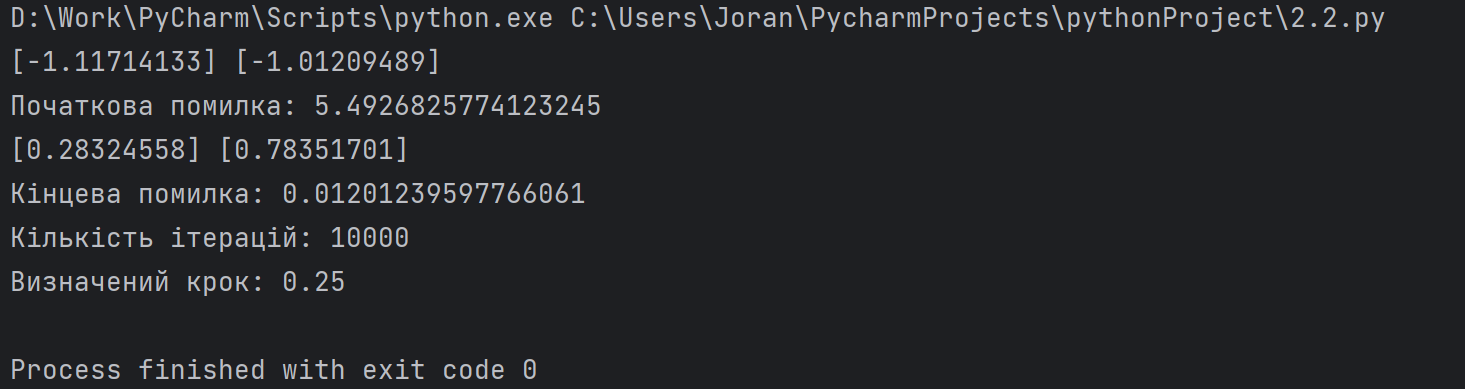
Для 100000 ітерацій, швидкість навчання 0.05



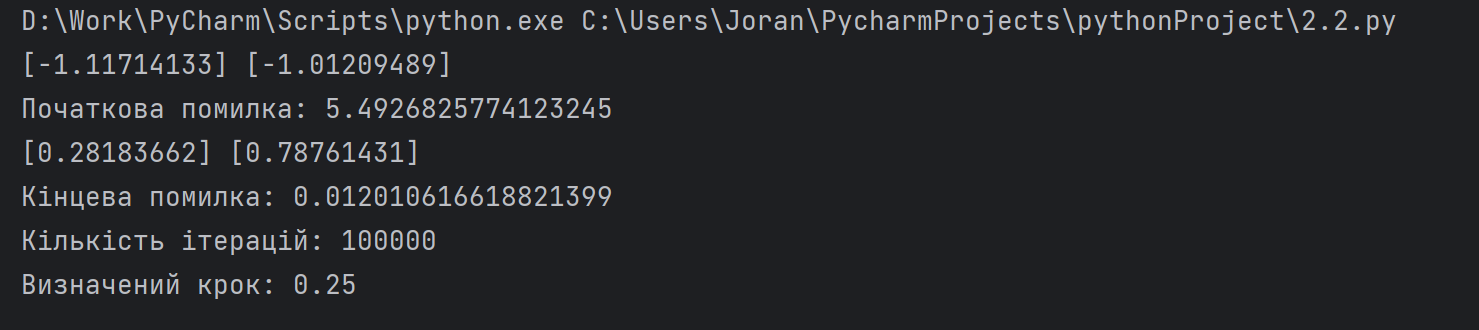
Для 1000 ітерацій, швидкість навчання 0.25

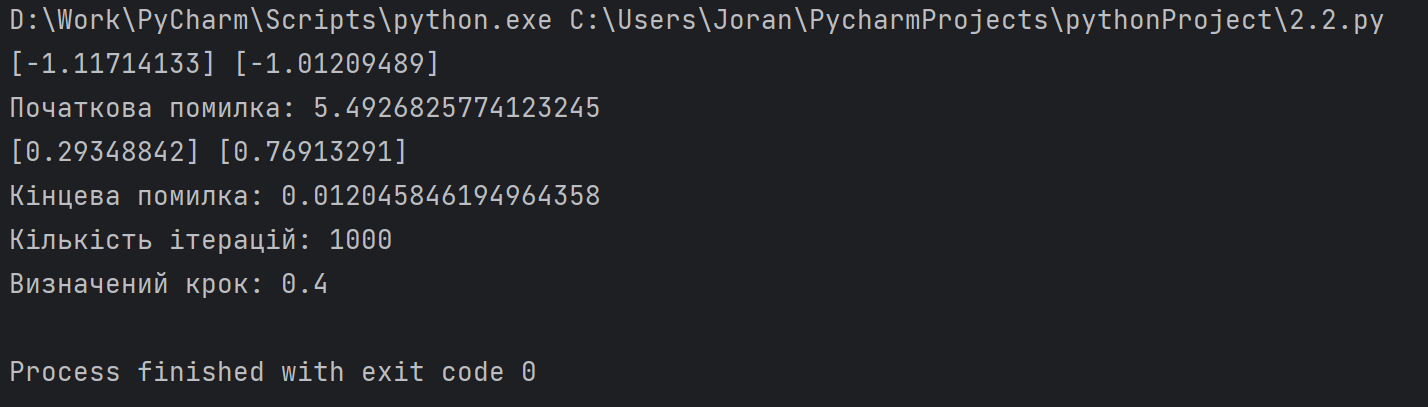


Для 10000 ітерацій, швидкість навчання 0.25

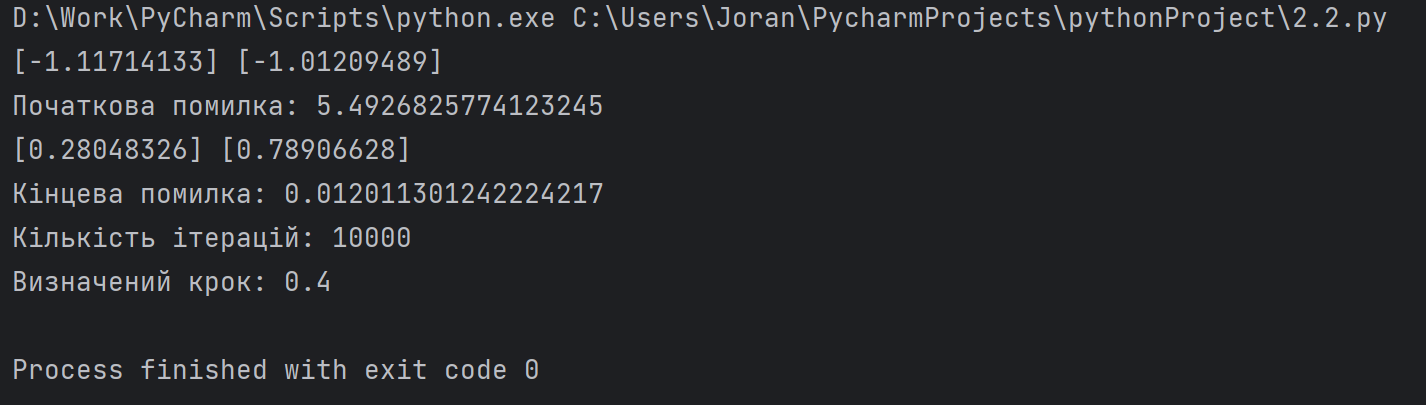


Для 100000 ітерацій, швидкість навчання 0.25

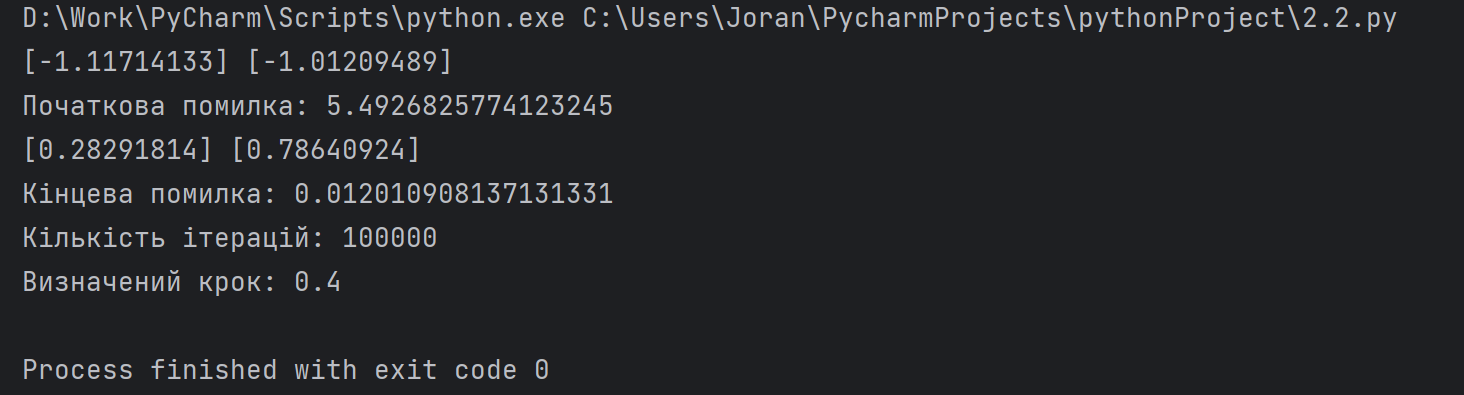


Для 1000 ітерацій, швидкість навчання 0.4 

Для 10000 ітерацій, швидкість навчання 0.4



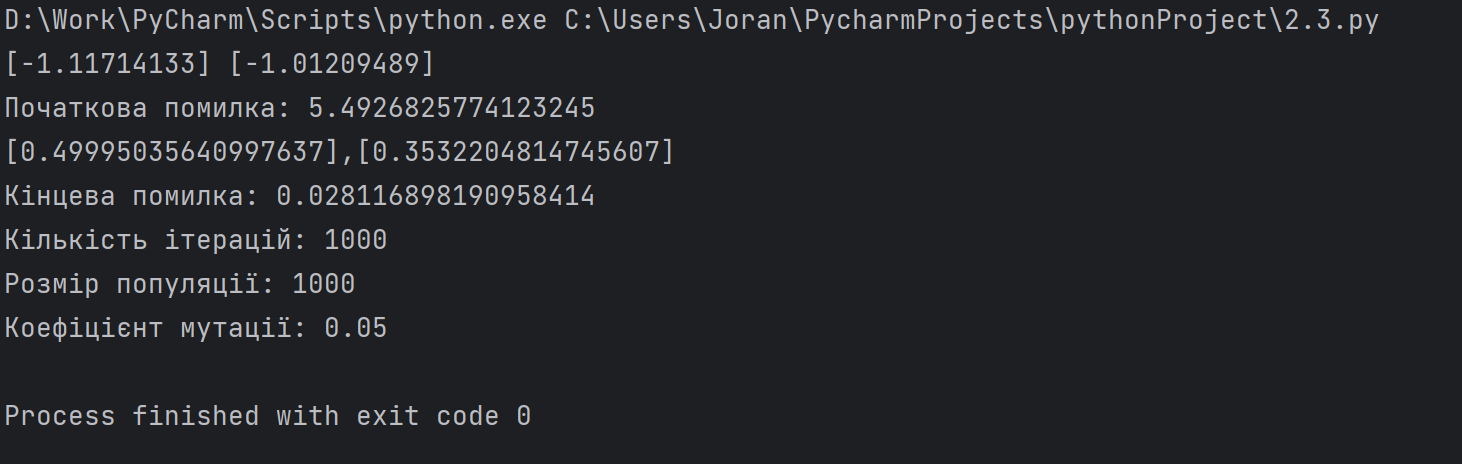
Для 100000 ітерацій, швидкість навчання 0.4



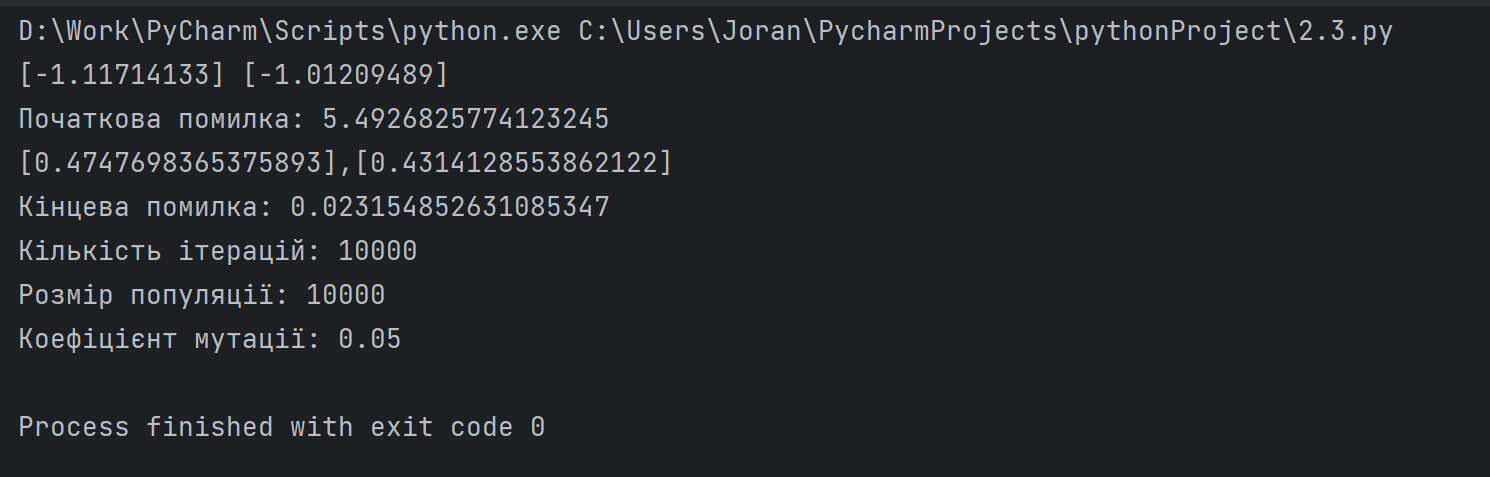
Зі збільшенням кількості ітерції та зменьшенням кроку ми отримаємо найменшу помилку. Найкращий отриманий результат 0.012010615212046332, який був отриманий кроком 0.25 та 100000 ітераціям

1. Генетичний алгоритм
2. import numpy as np  
   np.random.seed(42)  
   sz = 1000  
   x = np.random.rand(sz, 1)  
   y = x\*\*(1/2) + np.random.normal(loc=0.0, scale=0.1, size=x.shape)  
   idx = np.arange(sz)  
   np.random.shuffle(idx)  
   x\_train, y\_train = x[idx], y[idx]  
   # початкові випадкові значення для a і b  
   a\_initial = np.random.randn(1)  
   b\_initial = np.random.randn(1)  
   print(a\_initial, b\_initial)  
   # розрахунок початкової помилки  
   initial\_error = ((y\_train - (a\_initial + b\_initial \* x\_train)) \*\* 2).mean()  
   print(f"Початкова помилка: {initial\_error}")  
   population\_size = 10000  
   num\_generations = 1000  
   mutation\_rate = 0.1  
   # Створюємо початкову популяцію  
   population = np.random.randn(population\_size, 2)  
   for \_ in range(num\_generations):  
    # Обчислюємо помилку для кожного індивіда в популяції  
    errors = np.array([((y\_train - (a + b \* x\_train)) \*\* 2).mean() for a, b in population])  
    # Вибираємо найкращих індивідів (тих, у кого помилка найменша)  
    fitness\_scores = 1 / (1 + errors)  
    probs = fitness\_scores / fitness\_scores.sum()  
    selected = population[np.random.choice(np.arange(population\_size),size=population\_size, replace=True, p=probs)]  
    # Створюємо нове покоління через схрещування  
    pairs = selected[np.random.randint(0, population\_size, size=(population\_size, 2))]  
    new\_population = pairs.mean(axis=1)  
    # Застосовуємо мутації  
    mutations = (np.random.rand(population\_size, 2) - 0.5) \* mutation\_rate  
    new\_population += mutations  
    # Замінюємо стару популяцію новою  
    population = new\_population  
     
   # Вибираємо найкраще рішення з кінцевої популяції  
   best\_idx = np.argmin([((y\_train - (a + b \* x\_train)) \*\* 2).mean() for a, b in population])  
   best\_a, best\_b = population[best\_idx]  
   print(f'[{best\_a}],[{best\_b}]')  
   # розрахунок помилки після оптимізації  
   final\_error = ((y\_train - (best\_a + best\_b \* x\_train)) \*\* 2).mean()  
   print(f"Кінцева помилка: {final\_error}")  
   print(f"Кількість ітерацій: {num\_generations}")  
   print(f"Розмір популяції: {population\_size}")  
   print(f"Коефіцієнт мутації: {mutation\_rate}")

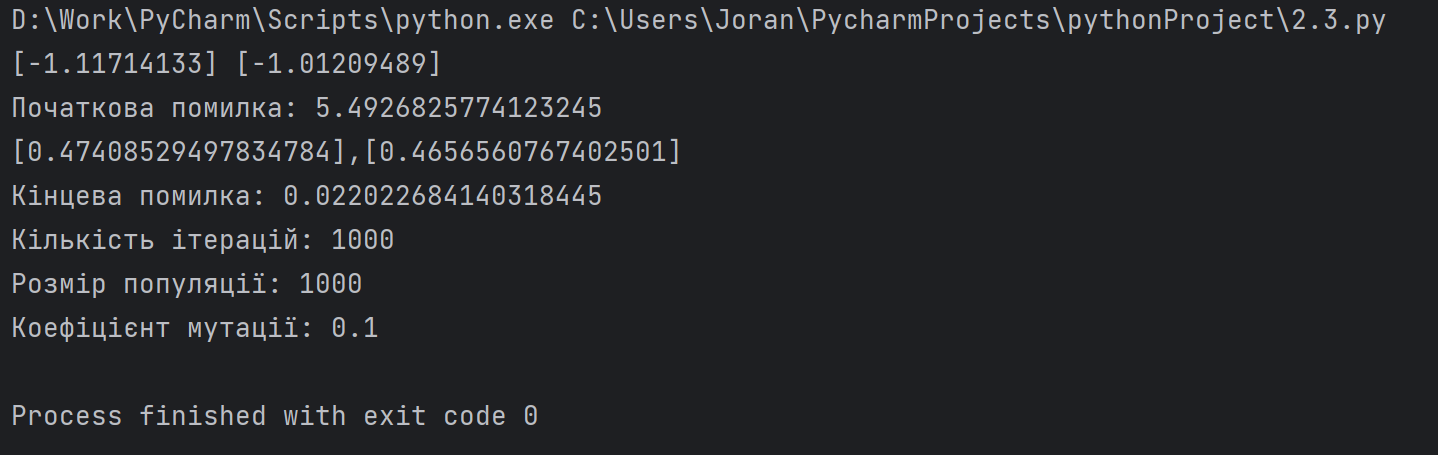
Для 1000 ітерацій, розмір популяції 1000, коефіцієнт 0.05



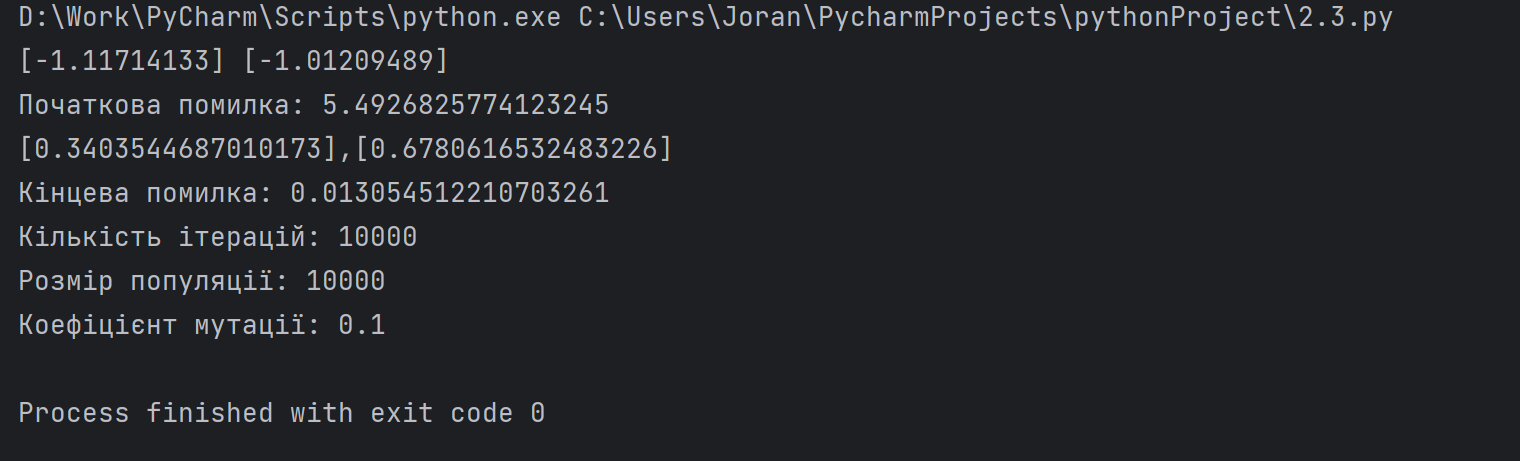
Для 10000 ітерацій, розмір популяції 1000, коефіцієнт 0.05



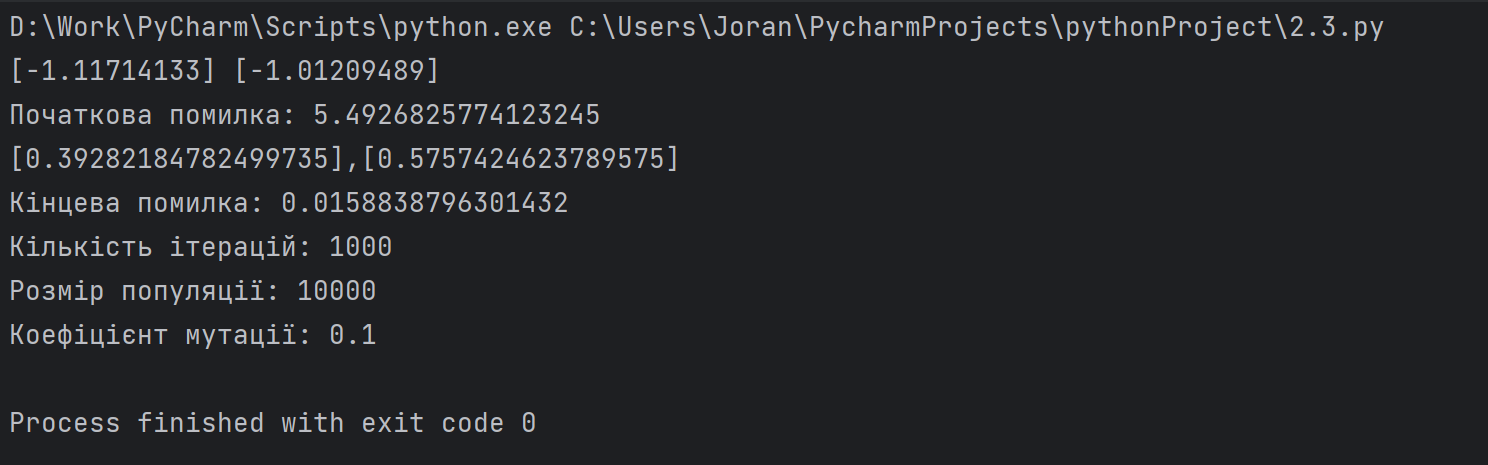
Для 1000 ітерацій, розмір популяції 1000, коефіцієнт 0.1



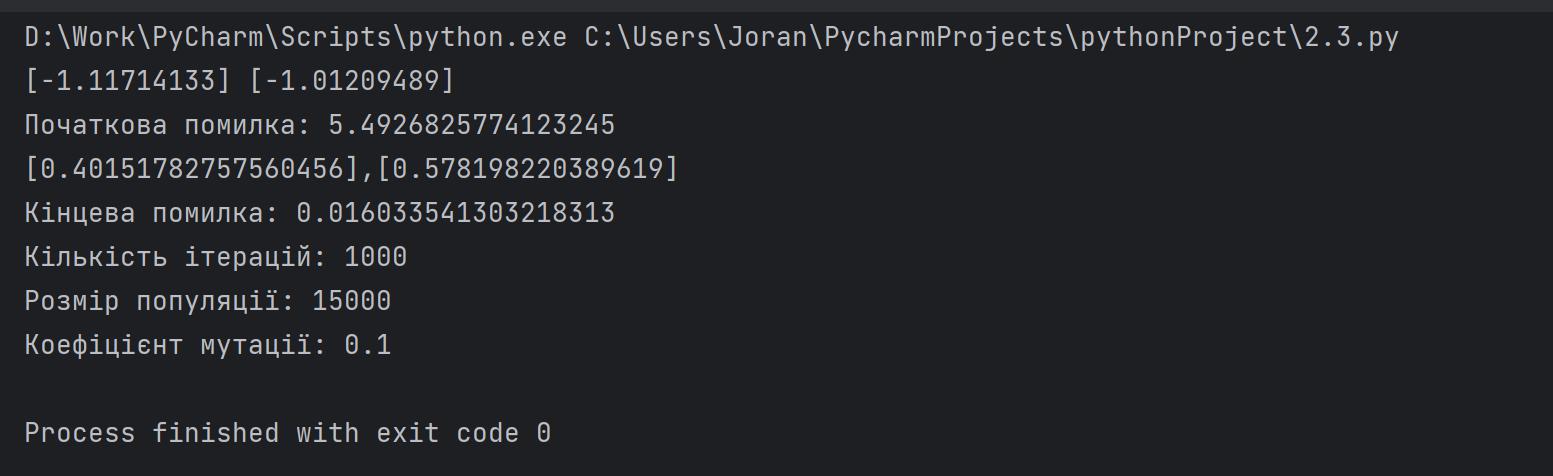
Для 10000 ітерацій, розмір популяції 1000, коефіцієнт 0.1



Для 1000 ітерацій, розмір популяції 10000, коефіцієнт 0.1

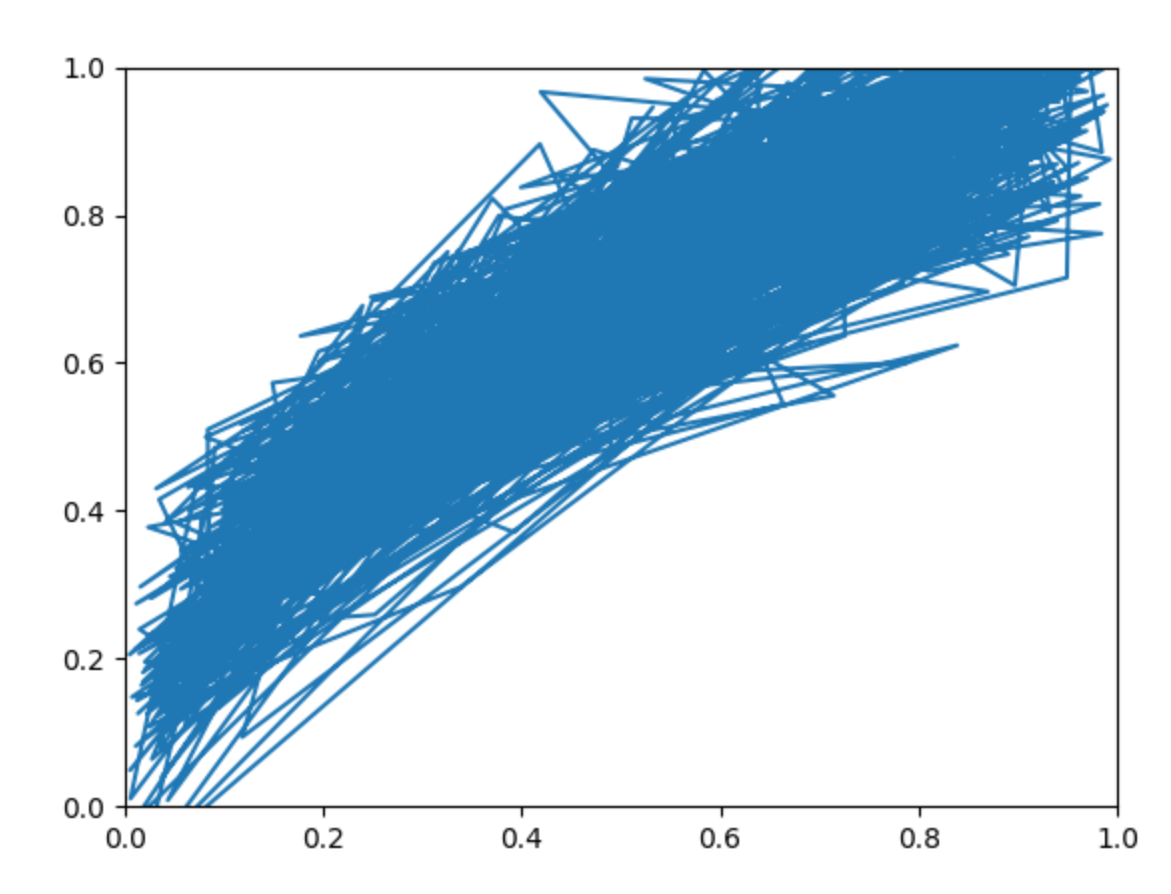


Для 10000 ітерацій, розмір популяції 10000, коефіцієнт 0.1



В результаті генетичного алгоритму найменшою кінцева помилка була для 10000 ітерацій, розмір популяції 1000, коефіцієнт 0.1 (0.0130541221073261)

**Графік функції з використанням випадкового шуму степеневої функції: y = x0.5 + noise**



# Аналіз роботи методів

Найменшу кінцеву помилку було отримано завдяки алгоритму градієнтного спуску (0.012010612927604842)

**Отримана точність:**

* Градієнтний спуск: Зазвичай має здатність до досягнення високої точності, особливо для гладких функцій. Точність зазвичай визначається налаштуванням швидкості навчання та кількістю ітерацій.
* Метод Hill Climbing: Може наближати локальний оптимум та мати обмежену точність через локальний характер алгоритму.
* Генетичний алгоритм: Може здати велику точність для глобальної оптимізації, але точність також залежить від налаштувань та популяції.

У розв’язаннях нашої задачі найточнішим виявился метод Методом Hill Climbing, який у результаті 1000 ітерацій видав найменший результат. Потім йде Градієнтний спуск і вже найбільший результат видав генетичний алгоритм за стандартним налаштуванням (розмір популяції = 100, мутація = 0.1). Загальна точність кожного методу також може залежати від якості початкових параметрів, налаштувань алгоритму, гладкості функції та інших факторів.

**Кількість ітерацій для отримання бажаної точності**:

* Градієнтний спуск: Кількість ітерацій може бути високою, особливо для складних функцій. Залежить від швидкості навчання
* Метод Hill Climbing: залежить від якості початкового стану.
* Генетичний алгоритм: залежить від числа поколінь, коефіціента мутації та розміру популяції.

**Необхідні ресурси:**

* Градієнтний спуск: Вимагає обчислення градієнта функції, а також велику кількість пам'яті для зберігання градієнтів.
* Метод Hill Climbing: Вимагає мало ресурсів, оскільки просто обчислює значення функції в околі поточної точки.
* Генетичний алгоритм: Потребує багато ресурсів для роботи з великим обсягом даних, обчислення фітнес-функцій та зберігання популяції.

**Залежність від старту:**

* Градієнтний спуск: Залежить від початкових параметрів, може покращувати результат з кожною новою ітерацією.
* Метод Hill Climbing: Сильно залежати від початкового стану.
* Генетичний алгоритм: Залежіть від початкової популяції, має можливість генерації нових рішень.

# Контрольні питання

1. **Що таке градієнтний спуск і для чого він використовується?**

Градієнтний спуск є одним із найпростіших і найбільш широко використовуваних алгоритмів оптимізації, який шукає мінімуми диференційованої функції. Ідея полягає в тому, щоб робити невеликі кроки в напрямку найшвидшого спадання функції, який задається градієнтом функції в даній точці. ін використовується для покращення продуктивності нейронної мережі шляхом налаштування параметрів мережі таким чином, щоб різниця між прогнозами мережі та фактичними/очікуваними значеннями мережі (що називаються втратою) була якомога меншою.

1. **Які основні складові градієнтного спуску?**

Ми маємо функцію втрат J(θ), де θ є вектором параметрів, які ми хочемо оптимізувати. На кожному кроці ми оновлюємо параметри за допомогою наступного правила: *θ = θ - α \** *J(θ),*

де α це розмір кроку або швидкість навчання, а J(θ) є градієнтом функції втрат в точці θ.

1. **Що таке швидкість навчання в контексті градієнтного спуску? Як вона впливає на процес оптимізації?**

Швидкість навчання в контексті градієнтного спуску - це величина, яка визначає, наскільки швидко алгоритм буде рухатися в напрямку мінімуму функції втрат. Чим більша швидкість навчання, тим швидше алгоритм буде наближатися до мінімуму, але також ймовірніше, що він застрягне в локальному мінімумі.

1. **Як градієнтний спуск впорядковує параметри моделі?**

Градієнтний спуск впорядковує параметри моделі, рухаючись в напрямку, протилежному градієнту функції втрат. Градієнт функції втрат вказує напрямок, в якому функція втрат найбільше зростає. Таким чином, рухаючись в напрямку, протилежному градієнту, ми зменшуємо значення функції втрат.Цей процес буде продовжуватися до тих пір, поки ми не досягнемо точки, в якій градієнт функції втрат дорівнює нулю. У цій точці модель буде мати оптимальні параметри для даного набору даних.

1. **Чому градієнтний спуск є одним з найпоширеніших методів оптимізації?**

Градієнтні методи є найпоширенішими в машинному навчанні з кількох причин:

* Обчислювальна ефективність: Обчислення градієнтів в багатовимірних просторах може бути відносно ефективним, особливо коли використовується автоматичне диференціювання, яке є вбудованим в багатьох бібліотеках машинного навчання.
* Збіжність: При певних умовах (наприклад, якщо функція втрат є випуклою та диференційованою) градієнтні методи гарантують збіжність до глобального оптимуму.
* Адаптивність: Варіанти градієнтного спуску, такі як AdaGrad, RMSProp та Adam, адаптують швидкість навчання в процесі оптимізації, що може допомогти уникнути проблем, пов'язаних з вибором швидкості навчання.
* Масштабованість: Методи, такі як стохастичний градієнтний спуск, добре масштабуються для великих наборів даних, оскільки вони не вимагають обчислення градієнту по всьому набору даних на кожному кроці.
* Нейронні мережі: Градієнтні методи є основою методу зворотного розповсюдження помилки (backpropagation), який є стандартним методом навчання нейронних мереж.

1. **Що таке метод Hill Climbing і як він відрізняється від градієнтного спуску?**

Метод Hill Climbing є ітеративним алгоритмом, який використовується для розв'язання задач оптимізації математичної функції. Він є одним з найдоступніших і простих методів пошуку локального максимума (або мінімуму) функції.

1. **Які основні кроки методу Hill Climbing?**

Спочатку вибирається випадкова точка в просторі рішень. Потім, вибирається сусідня точка і порівнюється значення функції в цій точці з попередньою. Якщо нове значення краще (більше для максимізації, менше для мінімізації), то переходимо в нову точку. Цей процес повторюється, поки не буде досягнуто максимуму або не буде виконано певна кількість ітерацій.

1. **Що таке генетичний алгоритм і в чому полягає його принцип роботи?**

Генетичний алгоритм — це евристичний алгоритм пошуку, використовуваний для рішення задач оптимізації і моделювання шляхом послідовного підбору, комбінування і варіації шуканих параметрів з використанням механізмів, що нагадують біологічну еволюцію. Є різновидом еволюційних обчислень (англ. evolutionary computation). Відмінною рисою генетичного алгоритму є акцент на використання оператора «схрещування», що робить операцію рекомбінації рішень-кандидатів, роль якої аналогічна ролі схрещування в живій природі.

1. **Чому деякі задачі вирішуються краще з використанням неградієнтних методів оптимізації?**

Деякі задачі вирішуються краще з використанням неградієнтних методів оптимізації через те, що цільові функції не є диференційовною. Градієнтні методи, які вимагають обчислення градієнта, не можуть бути застосовані до таких задач. Також якщо задача має багато локальних мінімумів, велику кількість параметрів та задача є обмеженою

1. **Як обрати метод оптимізації для конкретної задачі?**

Обирайте метод оптимізації залежно від вашої конкретної задачі, обмежень і доступних ресурсів. Іноді ефективним може бути використання комбінації різних методів або налаштування параметрів цих методів для досягнення найкращого результату.

* Градієнтний спуск:

Використовуйте градієнтний спуск, коли ви маєте диференційовну цільову функцію (яка може бути оптимізована за допомогою похідних). Помірно ефективний для задач з багатьма локальними мінімумами, тому може бути важко використовувати для несиметричних функцій втрат.

* Метод Hill Climbing:

Може бути корисним, коли вам потрібно знайти локальний максимум або мінімум. Однак метод Hill Climbing може застрягти в локальних оптимумах, і він не завжди знаходить найкращий глобальний оптимум.

* Генетичний алгоритм:

Генетичні алгоритми часто використовуються для глобальної оптимізації, особливо в задачах з багатьма локальними оптимумами.

Цей метод може бути корисним, коли ваша цільова функція не є диференційовною або не має аналітичної форми.

**Висновок :** ознайомилися із різноманітні методи оптимізації, які використовуються в машинному навчанні, зокрема, в області нейронних мереж.